Optimization of Monte Carlo simulations

Henrik Bryskhe
Abstract

Optimization of Monte Carlo simulations

Henrik Bryskhe

This thesis considers several different techniques for optimizing Monte Carlo simulations. The Monte Carlo system used is Penelope but most of the techniques are applicable to other systems. The two major techniques are the usage of the graphics card to do geometry calculations, and raytracing. Using graphics card provides a very efficient way to do fast ray and triangle intersections. Raytracing provides an approximation of Monte Carlo simulation but is much faster to perform. A program was also written in order to have a platform for Monte Carlo simulations where the different techniques were implemented and tested. The program also provides an overview of the simulation setup, were the user can easily verify that everything has been setup correctly. The thesis also covers an attempt to rewrite Penelope from FORTRAN to C. The new version is significantly faster and can be used on more systems. A distribution package was also added to the new Penelope version. Since Monte Carlo simulations are easily distributed, running this type of simulations on ten computers yields ten times the speedup. Combining the different techniques in the platform provides an easy to use and at the same time efficient way of performing Monte Carlo simulations.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction to Monte Carlo simulation with Penelope</td>
<td>2</td>
</tr>
<tr>
<td>1.1</td>
<td>Photon interactions</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Electron and positron interactions</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Penelope</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Phase Space</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>Monte Carlo Problems</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Particle Tracer – A M.C. simulation development platform</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>Introduction</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Implementation</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1</td>
<td>File format</td>
<td>9</td>
</tr>
<tr>
<td>2.2.2</td>
<td>The XML parser eXpat</td>
<td>10</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Data representation and functionality</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Ray tracing simulation</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>The Method</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>Computational problems</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>A complete rewrite of Penelope from FORTRAN to C</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>Porting considerations and implementation</td>
<td>16</td>
</tr>
<tr>
<td>4.3</td>
<td>Verification</td>
<td>19</td>
</tr>
<tr>
<td>4.4</td>
<td>Optimizations</td>
<td>19</td>
</tr>
<tr>
<td>4.4.1</td>
<td>The Random function</td>
<td>20</td>
</tr>
<tr>
<td>4.4.2</td>
<td>The secondary particles stack procedure</td>
<td>20</td>
</tr>
<tr>
<td>4.5</td>
<td>Results</td>
<td>20</td>
</tr>
<tr>
<td>4.6</td>
<td>Distribution</td>
<td>20</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Structure</td>
<td>21</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Slave considerations</td>
<td>21</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Distribution testing</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>GPU programming</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>History and Introduction</td>
<td>23</td>
</tr>
<tr>
<td>5.2</td>
<td>Programmable Vertex and Fragment shaders</td>
<td>25</td>
</tr>
<tr>
<td>5.3</td>
<td>GPGPU</td>
<td>26</td>
</tr>
<tr>
<td>5.4</td>
<td>Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>5.5</td>
<td>Optimisation</td>
<td>29</td>
</tr>
<tr>
<td>5.6</td>
<td>Results</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>31</td>
</tr>
<tr>
<td>7</td>
<td>Appendix A – A fast ray / triangle intersection algorithm</td>
<td>32</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>32</td>
</tr>
<tr>
<td>7.2</td>
<td>Algorithm</td>
<td>32</td>
</tr>
<tr>
<td>7.3</td>
<td>Code</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>References</td>
<td>35</td>
</tr>
</tbody>
</table>
1 Introduction to Monte Carlo simulation with Penelope

Monte Carlo simulation is a tool that can help both physicists and engineers to understand how radiation works in different environments. This can be a difficult task if the geometry is not simple or symmetrical.

If you would like to simulate how much energy is deposited in a region at some distance from a radioactive source there are different ways to do this.

Figure 1-1: A typical Monte Carlo simulation problem. Particles emanate from the source and then enter a body. There they interact and deposit energy inside it. Typically you are only interested in the distribution in a specific region inside the body.

If the geometry is simple you can solve it using the Boltzmann transport equation. This is a direct approach and it means that you need to solve a partial differential equation (PDE) with boundary conditions. However in real life the geometry is not simple resulting in complex boundary conditions. This makes it necessary to find another approach. The problem is to find the particles that emanate from the source and travels to the region of interest, and find out how much energy they deposit there. They deposit energy due to different interaction between the particles and the atoms/molecules.

One way to solve this is to generate particles at the source. The particles should have an initial position, direction and energy. Then you track each single particle as it travels through different materials and undergoes different interactions. If you are lucky the particle will enter the region of interest and deposit energy there.

This is how Monte Carlo simulation works. The name Monte Carlo makes you think about gambling because of the casino in Monaco. The connection to gambling is that during the particle transportation random numbers are used in order to calculate if an interaction should occur. The Monte Carlo Simulation (MCS) program calculates the probability of different interactions between a particle and the material it travels in. These probabilities are called Differential Cross Sections (DCS).

The main problem with MCS is that you need the dose distribution to converge. The reason is that when one particle deposits energy you have no idea how common it is for particles to deposit dose with certain energy at a certain position. Therefore you will need to keep track of the variance of the deposited dose during the simulation. Then you are able to simulate until the variance has converged to an acceptable value. It usually takes a lot of time depending on the problem and geometry as most particles sent out from the source will not deposit energy in the region of interest.

This makes it necessary to change parameters in the simulation to gain speedup without changing the result. The technique is called variance reduction. For instance, in the example Figure 1-1, it is not necessary to send out particles in all directions from the source. We could limit the initial direction of the particles towards the object or the region of interest. But if we limit the direction too much we will get a different result because of the lack of particles in some directions. Particles also scatter inside objects and change direction. If you limit the
direction you might miss the contributions they make. This is shown in Figure 1-2 and Figure 1-3.

![Particle interaction diagram]

**Figure 1-2:** Particle emanated from the source and enters the blue object. An interaction occurs which makes the particle leave in a completely different direction.

However, we need to be careful when we apply variance reduction. The reason is that we can end up with a different result if we apply wrong parameters. It is often required to use variance reduction because of computer power limitations.

![Diagram of particle interaction and energy deposition]

**Figure 1-3:** This describes Figure 1-2 in detail. A particle leaves the source and after an interaction enters the region of interest (ROI), where it deposit energy.

The MCS system changes the DCS depending on what material is used during the simulation. Therefore, probabilities changes for different interactions. If the particle is located inside lead or tungsten it is more likely that interactions will occur more often than in air. The mean free path describes the mean path length a particle will travel without an interaction occurring. The mean free path varies depending on the atomic and molecular structure of the material and the energy of the particle. The precision and quality of a MCS system depends on how good DCS it can generate for different materials and energies.

Usually a MCS system tracks three different particles, namely photons, electrons and positrons and their interactions. There exist different interactions for each type of particle.

### 1.1 Photon interactions

- **Rayleigh scattering**
  
  This process occurs when incident photons are scattered by bound electrons without exciting the atom. The energy of the incident and scattered photons are the same.
• **Photoelectric absorption**
  When incident photons interact with an outer shell electron, all of its energy is absorbed by the electron that escapes its shell. If the electron was located in an inner shell another electron from an outer shell will replace it and this will release energy $U_i$, called the characteristic radiation.

\[ E_e = E - U_i \]

• **Pair production**
  A travelling photon gets close to the nucleus and the energy is greater than the rest mass of an electron and a positron ($E > 1.022\text{MeV}$). The photons energy is absorbed by the pair production process and is shared between the electron and positron that escapes.

• **Compton scattering**
  A photon with energy $E$ interacts with an atom. The atom absorbs the photon and re-emits another one with the energy $E'$. After the Compton interaction the target electron, with which the incoming photon interacted, is knocked into a free state leaving the atom in an excited state.
1.2 Electron and positron interactions

- **Elastic collisions**
  The initial and final states of the atom are the same in elastic collisions. The incoming electron or positron is deflected a certain angle but does not change state. It loses a small amount of energy but the quantity is so small that it can be neglected. The angular deflection of the electron trajectories is mainly due to the elastic collisions.

- **Inelastic collisions**
  For the inelastic collision the incoming electron or positron loses some of its energy. The reason is that it will leave the atom in an excited state or as an ion. For low or intermediate energies this is the dominant factor for energy deposition.
• **Bremsstrahlung emission**
  A particle that is accelerated emanates electromagnetic radiation. Therefore when an electron or positron is deflected around a nucleus it will emanate radiation due to acceleration that takes place.

![Bremsstrahlung](image1)

• **Positron annihilation**
  A positron can annihilate an electron in the current media and thus two photons are created with the same energy $E$.

![Positron annihilation](image2)

### 1.3 Penelope

Penelope is a Monte Carlo simulation library developed at the University of Barcelona. The material library that comes with Penelope makes it possible to calculate the DCS for particle interactions with energies from a hundred eV to 1 GeV. A geometry package is also included that is called PenGeom. This package makes it possible to track the particles trajectories inside geometries consisting of quadrics. A quadric is a general second degree surface. Surfaces that can be represented as quadrics are spheres, cylinders and planes etc. PenGeom also supports Constructive Solid Geometries (CSG) which makes it possible to use the mathematical operations Union, Intersection and Difference between the quadrics in order to create new surfaces.

Penelope in itself is not a simulation program that can be used for simulations. It merely provides the code necessary to simulate a particle inside materials. It’s up to the user to create the code for the sources, where it should store energy deposited etc. Therefore, it takes some time to create a new simulation program for a specific simulation problem. The new code has to be tested and verified and that can also take significant time.

### 1.4 Phase Space

A phase space (PS) is a simple tool in the Monte Carlo system that can provide vital information about the simulation. Recording only information about the deposited dose and the variance will only answer the question of how much radiation will reach a region from the source. It will not answer how and where the particles travelled on their way. Sometimes this information could be as important as the other.
The phase spaces stores information about each particle passing through it, energy, position, direction, particle type etc. Additional information such as where the particle has been or interacted could also be stored in the PS.

Analyzing the PS will provide information on where particles have been and this is vital in some cases. For instance, if you get unwanted peak values and you analyze the phase space, you might notice that the particles that entered a certain object give rise to these peaks. Changing the geometry or adding additional shielding can reduce the peaks. A new simulation with a new geometry can be performed, and in that simulation we can study if the peaks are gone.

Saving a PS to a file could also greatly reduce the amount of simulation time. If the particles trajectories is traced only half the way to the ROI and stored on a PS, then it is possible to continue the simulation from the PS. However, in a simulation with similar geometry, only the last part of the geometry has changed, then it is possible to start this simulation from the phase space as they share the first part of the geometry. Thus, the time to simulate the first part of the geometry can be avoided for the second simulation. Simulating a part of the problem is quite common and it can be easier to analyze a smaller part of the problem where the particles have not travelled so far.

1.5 Monte Carlo Problems

A typical Monte Carlo simulation problem is when the user needs to know how much energy is deposited in a certain region. The reasons for this could be manifold. For instance it could be a doctor that is going to do a treatment of a patient with radiation therapy. The patient will be radiated close to a vital organ. Then the doctor need to know before the treatment how much radiation he can expose the patient to, without endangering the patients vital organs.

Another example is when a new radiation shield is going to be designed. It is possible to use MCS to see how much radiation will pass through the new shield without making a prototype. This could avoid potential problems with a shield that is dimensioned too small and provide information about it in time.

Most importantly, it can be used in order to investigate the radiation transport in a complex system. It may not always be an easy task to understand why certain dose peaks exist. Is there a natural explanation for the peaks or could it be leakages in the system. MCS can provide the relevant information to investigate peaks and understand their nature. A key tool for investigating this is the phase spaces.
2 Particle Tracer – A M.C. simulation development platform

2.1 Introduction

A platform was needed in order to try out different implementations of Monte Carlo simulation speedup techniques. It is impossible to know all different kinds of techniques you might wish to try out. Therefore, it is important to write the code as general as possible. Keeping this in mind, a tool called Particle Tracer was created. It was written in C++ to take advantage of the object oriented programming capabilities it provides. The core of Particle Tracer was written in ANSI/ISO\(^1\) standard C++ to make it available on most computers and operating systems. On top of the core a GUI\(^2\) were created using MFC\(^3\). The reason for using a GUI is to be able to visualize the system completely with sources and dose distribution results. It is important to get an overview of everything and verify the setup as a simulation could run for days even on a cluster of computers. If the setup of the system is wrong you might not notice it until after the simulation is finished and you do not have the desired result. An important feature in the program is to support loading of CAD geometry. \textit{Figure 2.1} shows an example of this. This means that the program should be able to run simulations with models created with a CAD program.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{cad_model.png}
\caption{CAD modelled Radiation Unit from Leksell Gamma Knife 4C rendered with active sources in ParticleTracer.}
\end{figure}

\begin{footnotesize}
\begin{enumerate}
\item ANSI/ISO is standard for compiler creators to follow that makes code portable from one compiler to another.
\item GUI – Graphical User Interface. A window / dialog based interface to a program. Used to create user friendly interfaces to programs.
\item MFC – Microsoft Foundation Classes. A class library to create windows programs easily.
\end{enumerate}
\end{footnotesize}
2.2 Implementation

2.2.1 File format

Careful investigation was done in order to know which file format to support. The aim with Particle Tracer was to create a platform that could be used in the unforeseen future, so that future implementations can be implemented without rewriting the code. The file format represents the triangles from the CAD model. Hence, choosing the file format is very important if you wish to keep as much information as possible from the CAD model. For instance, one possibility is to use a raw format. The CAD program would then convert the model into triangles and save them in a large list. A problem occurs when you wish to know which triangle belongs to a certain part of the CAD model. Therefore, a file format that keeps track of the model parts where chosen, namely the XGL format. Another advantage with the XGL format is that it keeps track of the original parts along with transformation matrices. That means that parts that are identical, except for their transformations, only exist at one place in the memory along with all the corresponding transformations. See Figure 2-2.

This is only useful if you have symmetry in your model. With this information it is only necessary to do space partitioning on the original part. Since it exist only one copy of the original object triangles they can be further subdivided when performing a subdivision. That will yield a performance gain when testing for different types of intersections. This is of course true under the assumption that there exist multiple copies of objects in the CAD model. If it exist only one copy of each object you would loose performance due to the fact that for each operation that involves the original object a change of base transformation has to be performed. If, on the other hand, all the transformed object’s triangles where saved, then the subdivision of all these triangles would require more memory.

![Figure 2-2: Only the untransformed original object triangles exist in memory. Then multiple copies exist with individual transformations.](image)

Hence a decision had to be made on which method to support. Due to the fact that the CAD models that were going to be used in the simulations were highly symmetrical the choice was easy. The change of base transformations are not time consuming, only a couple of multiplications and additions are required. Only the scaling, rotation and translation transformations are used with Homogenous coordinates, the matrices and vectors have the following form:

\[
M = \begin{bmatrix}
    a_{11} & a_{12} & a_{13} & t_1 \\
    a_{21} & a_{22} & a_{23} & t_2 \\
    a_{31} & a_{32} & a_{33} & t_3 \\
    0      & 0      & 0      & 1
\end{bmatrix}, \quad P = \begin{bmatrix}
    x_1 \\
    y_1 \\
    z_1 \\
    1
\end{bmatrix}
\]
To transform a point \( P \) with matrix \( M \) requires only \( 4 \times 3 = 12 \) multiplications and \( 3 \times 3 = 9 \) additions. Note as the homogenous coordinate \( w = 1 \) at all times we can simplify the calculations and no projection division is required.

Both \( M \) and its inverse are pre-calculated at the initialization stage for all objects in order not to waste the CPU at runtime. However, as mentioned above, if all the object’s triangles are stored in their transformed positions the change of base transformation is unnecessary and CPU power is saved.

### 2.2.2 The XML parser eXpat

An XGL file is a XML document, as XGL is a part of XML. Thus, to read a XGL file a XML parser is needed. XML is also a well established standard and XML documents are text files using Unicode. Thus, you can transfer XML files between different systems without problems.

The XML parser eXpat were chosen as it is open source and free to use. It also has support for loading a small chunk of a larger file. This could be useful since larger CAD models will be very large due to the fact that they are text files. Expat is also very fast and is used in many other applications. For example, expat is being used in the open source project Mozilla.

### 2.2.3 Data representation and functionality

When the XGL file data is read it needs to be stored in the memory. This can be done in many different ways, and the way you choose is a critical issue. The simulation speed will greatly depend on how you store your data in memory and even more, you may find it impossible to implement future improvements depending on the choices you made. For instance, to speedup computations a graphic card could be used. But a graphic card will require that triangles are sequential and aligned in memory. If that is not the case the graphic card can not use its great computational power or at least you will not get the peak performance from it. Another advantage with sequential data in memory is that cache misses are greatly reduced.

Taking advantage of object oriented program and utilizing virtual functions the functionality of ParticleTracer(PT) is easy to extend. This will make it easy for other programmers to implement new functionality as well. Again, a need can be seen for a well managed and structured data representation. If other people will potentially add new functionality the structure need to be intuitive and easy to grasp. Figure 2-3 shows the structure outline for the geometric representation.

The whole PT is written using type defined variables for the calculations and data representation. This makes it possible to run PT using only single precision or double precision data. Even a mixture between them is possible. For instance the data can be stored on a media in single precision even though all calculation is performed in double precision. When the data is loaded it is then converted back to double precision. The main reason for

![Figure 2-3: Outline of the geometric data representation in ParticleTracer. Some of the geometries could be handled by the CPU or the GPU. For improving the intersection calculations etc.](image.png)
this is to reduce data storage. Neither Intel or AMD support for single precision calls to the FPU functions, such as exp(), sin() and cos(), therefore no computational performance is gained. Note, however, that in some cases an increase of the overall speed up could be expected if the data is stored in single precision. This is due to the fact that a fetch instruction will be able to retrieve two single precision values at the same time as a double precision value. However, the CPU will have to convert these values from single precision (SP) to double precision (DP) when a call to sin() is made. Then, it has to convert them back to SP. Hence, programs that do a lot of memory reading/writing and not use mathematical functions will gain a lot from using SP.
3 Ray tracing simulation

3.1 Introduction

Since a normal Monte Carlo simulation is very time consuming and the simulation time depends on the complexity of the geometry, a new method were developed to obtain results faster. The method is intended to be used so that an approximation of the dose distribution in space can be obtained efficiently, even for complex geometries. It is based on a normal attenuation algorithm and computes the dose distribution in a point. The algorithm first calculates how much material a ray passes through from a source to a point in space. Then it applies the attenuation method and calculates the dose at the point. We simply call it ray tracing simulation.

3.2 The Method

As mentioned above the ray tracing method is based on a linear attenuation formula, and its purpose is to calculate the dose distribution in a point. The method will not consider any secondary particles at all.

This is the attenuation formula:

\[
D_p = D_c \left( \frac{R_1}{L_1} \right)^2 e^{-\sum_{i=1}^{n} l_i \mu_i}
\]

- \(D_p\) – Target dose rate at the point of measurement (p).
- \(D_c\) – Measured dose rate at the point of measurement (c).
- \(R_1\) – Distance between source and point of measurement (c).
- \(L_1\) – Distance between source and point of measurement (p).
- \(l_i\) – Thickness of material i.
- \(\mu_i\) – Coefficient of attenuation for material i.
- \(n\) – Number of materials.
As it is difficult to measure the dose rate at the source it is instead measured in a water phantom at point c. By using the amount of dose that has been absorbed by the source, it is possible to calculate backwards the absorbed dose. Hence by adding this absorption the dose rate at the source will be known. The terms $R_1$, $l_w$ and $\mu_w$ in the equation comes from this.

The next step is that the program calculates the amount of material a ray from the source point to a point p intersects. What material each model part consist of together with each material attenuation coefficient must be known. Also the distance between the source and the point of measure must be provided. The attenuation coefficients are provided by the user as well as the sources and the points of measure.

The program assumes that both the point of measure and the specified source is located outside any material. Then the algorithm can make the assumption that when it checks for intersections it will always end up with an even number of intersections.

![Figure 3-1: Even and odd number of intersections. If a segment is intersected with an object and the number of intersections is even then the segment starting point and ending point is outside the object or both points where inside the object. Odd means that one point is located inside and the other outside.](image)

The assumption is needed to make the algorithm faster. It would require much more computational time if the assumption is not made. When an odd number of intersections occur the algorithm needs to know if it is the start or the endpoint that is located inside the object. The reason is that it needs this information to calculate the amount of material it intersects correctly. See Figure 3-1 for the different cases.

The algorithm is implemented in ParticleTracer (PT). Within PT each object is assigned a material with a coefficient of attenuation. Setting up sources and destinations is also done from PT. See Figure 3-2 where PT has been setup with one destination point, located at 1.0m at x-axis, and all the LGK sources are active. Both an opaque and transparent picture can be seen.

The method should do the job fast even for geometries with great complexities. The algorithm first checks for intersection against a bounding box of the object. If it intersects the bounding box it checks for intersection against all the triangles.

What you are able to do now is to produce maps of the dose distribution in space and get the results quite fast. The PT has a feature where it is able to display the results using a color map. See Figure 3-3 where the result of a simulation is shown.
Figure 3-2: The dose rate is to be calculated at the point 1m along the x-axis outside the LGK 4C. The machine has 201 radioactive sources. Right image is with opaque geometry and the left with transparent.

Figure 3-3: The dose distribution with a color map that maps low dose values with blue and high with red and white.

3.3 Computational problems

Two major problems occurred during measurements. The first is that the program can not always calculate intersections correctly when the ray is parallel to the surface and at the same time very close to the surface. Sometimes the intersection algorithm reports the wrong number of intersections.

These problems only occurred when testing that the algorithm calculated the intersections correctly. We tested rays along the coordinate axis and some geometry had parallel surfaces along the axis and the program failed in these cases. However the CAD program SolidEdge, which we used to export the CAD model, also failed to calculate the intersections correctly. This is because of the finite precision in the computer. By using double instead of single precision the program works better.

During all our “real” measurements we never experienced any such troubles. The reason is that the spatial locations of the sources are chosen in a way such that they never will end up parallel to a surface.

The other problem was the triangulation of the surfaces. At surfaces with high curvature rays can escape through the gaps that occur due to the error in the triangulation. This gives
extremely high dose rates in a very small area. However, it could as well be a natural leakage in the geometry.

To remedy both of these problems the ray tracing program was enhanced with features that makes it possible to produce lists of the sources contributing the largest dose in a measure point. It is then possible to visualize the ray from the source that gave the highest dose contribution. It can be easily seen if the ray is passing through a gap or is parallel to a surface.
A complete rewrite of Penelope from FORTRAN to C

4.1 Introduction
The physics library Penelope is the real bottleneck of the Monte Carlo simulation. If removing all of the geometrical computation, you are left with what the physics core requires. Hence an optimization of Penelope would reduce the overall simulation time. The physics library provides the code for the particle interactions. Parameters that have to be known are what kind of material the particle resides in and the energy. From this information Penelope calculates the probabilities for an interaction and then lets the main program know what kind of interaction occurred. Penelope keeps track of all the secondary particles that where created during an event by placing them on a stack. The main program is then responsible for retrieving the secondary particles from the stack and continuing simulating them.

4.2 Porting considerations and implementation
Penelope was originally written in FORTRAN. A natural selection of a new language for translation would be C. C compilers are available on almost all platforms and thus if Penelope would be written in ANSI C you would be able to compile and run it on almost every computer. For instance, it could be really interesting to run a simulation on a PowerPC.

One question that might be asked is, “Why not write in C++ instead of C?” In C++ it is easier to maintain the code and develop it further. However, some developers use C and they will not be able to make use of a C++ port. But the C++ developers can utilize the C port since C is subset of C++. If there is a need for a C++ port of Penelope it is a lot easier to create it from the C port. For instance, the people in Barcelona (who originally developed Penelope) might find it attractive to continue their future work of Penelope using a C++ port.

A PowerPC is more suited than an ordinary PC for this kind of program. It provides more registers and faster buses. Especially it would be interesting to compare simulations based on single and double precision. An ordinary PC (Intel, AMD) does not support single precision calls for the mathematical function, such as sin, cos and exp. Hence, if you compile the code using single precision you would still need to convert variables into double precision when you call the math functions resulting in increased CPU time. But the PowerPC supports these functions which makes these conversions obsolete.

The porting job was split into two phases. The first phase was to rewrite the code from FORTRAN to C, and the second phase was to confirm that the C code was correctly ported. Careful consideration was made in order to create code that would compile on different platforms. You might wish to compile the code on a Macintosh using OS X or a UNIX machine using Solaris. That means that the code should be able to compile using both 32-bit and 64-bit code and using single or double precision calculations as well. To solve this problem, preprocessor macros where used to define definition and mathematical functions and variable types. Hence the type definition Real where introduced and only by changing the code at one place it can be compiled with single or double precision that works under 32 or 64bits systems.

Furthermore, all of the new Penelope code was to be written in the C ANSI standard. It makes the program able to compile on all compilers that supports this standard. In turn it makes Penelope available on all platforms that have an available C ANSI compiler.

An important functionality was to make PenelopeC fully reentrant. The reason for having an application that is fully reentrant is that it is thread safe. That means that you can start two or
more threads from the main process without running into problems. In order to do this there can not be any global variables at all.

To create a fully reentrant program, contexts where created for the different parts. The contexts are structures containing all the data needed for the each part. For the Penelope core a context called PenelopeContext where created which had the structure described in Listing 4-1.

```c
typedef struct _PenelopeContext
{
    int materialCount;

    // Penelope kernel buffers
    CBRAng cbrang;
    CEBR ceb;
    CEBR01 ceb01;
    CEEL00 ceel00;
    CEGrid cegrid;
    .
    .
    CRelax crelax;
    CAData cadata;

    struct
    {
        int iseed1;
        int iseed2;
    }rseed;

    // Simulation buffers
    Track track;
}PenelopeContext, *PPenelopeContext;
```

Listing 4-1: The Penelope context.

The geometry, simulation parameters and distribution part got similar contexts.

Macro definitions where used to make the porting easier. The reason for using these macros was to make the C code similar to the FORTRAN code. FORTRAN is using common blocks to share data between functions and these blocks have names. The macros were given names from their common block and variable name. Some examples of macros are shown in Listing 4-2 below:

```c
typedef struct _CHIST
{
    int ilba[5];
    int iza;
    int isa;
}CHIST, *PCHIST;

#define _CHIST_ILBA(i)  ctxG>chist.ilba[(i)]
#define _CHIST_IZA       ctxG>chist.iza
#define _CHIST_ISA       ctxG>chist.isa
```

Listing 4-2: Example of C macros defined for the common block CHIST.

By using these macros the FORTRAN and C code would have similar syntax and thus making the port job easier. The preprocessor compiler can compile the code without the macros and produce the expanded code. Hence the macros can be removed from the code if necessary.

The FORTRAN code consisted of a lot of nested goto branches, and the code optimizer does not work well with such branches. Hence, all the goto-branches where removed and replaced with while- and for loops. At some points these replacements required the insertion of an extra variable but mostly there was no need for that. The code is also a lot easier to read without the goto branches, which is important if someone wants to read and understand it.
All the FORTRAN code had arrays that were one-indexed. C usually uses zero-indexed arrays. One could do with one-indexed arrays but that would consume more memory and thus provide more potential cache misses. A big effort was put into making all these one-indexed arrays into zero-indexed. Problems occurred since some variables contained index values (which were now zero based) and others stored quantitative numbers (which would still have same values as the FORTRAN code). When variables could not be determined whether they were numbers or indexes, problems were close at hand. At some points in the code the same variable could be used as an index and a number. This stress that a careful verification of the code should be performed in order to remove the mistakes that had been made.

Another thing to improve was to support a variable number of materials. The original code only supports a fixed number of materials. That means that if you need to do a simulation with more materials than the program supports you need to recompile the program with increased maximum number of materials. If you run a simulation with a number of materials less than what the program supports it will allocate more memory than is needed. This could result in more unnecessary cache misses. With the C heap memory manager it is possible to support a variable number of materials. Hence when the program runs it will allocate the right amount of memory needed for the simulation.

There is a drawback with using a variable number of materials, namely that the application needs to use indirect memory addressing. Indirect memory addressing is slower than direct addressing because of the extra lookup as shown in Figure 4-1.

![Figure 4-1: Example of direct/indirect memory addressing. The variable y from the struct x is being accessed.](image)

A test program was created to show the differences between indirect and direct address which consisted of only reading and writing to the memory. The difference was as high as 60%. However, using direct addressing instead might not give a noticeable effect in Penelope since computations could be the cause for almost all CPU time. By changing only in the PenelopeContext one can make Penelope use direct addressing.
4.3 Verification

Since it is impossible for a human being to write larger code fragments without making mistakes it is very important to verify the code and correct the errors. The compiler takes care of all the syntax errors but some mistakes made, such as indexing errors, the compiler will not complain about. These mistakes can result in a particle getting the wrong energy or a particle interaction that should not occur. It can be very tedious to find these coding mistakes.

For that matter a log was created that for each particle interaction stored the information about its energy, position, direction and type of interaction that occurred. This log was implemented both in FORTRAN and C code. The idea was to check the two logs created from both programs and find where they differed. If the two logs where different we could trace which interaction that went wrong. Knowing this, it is possible to run the debugger there. Then by running the FORTRAN debugger at the same time as the C debugger stepping forward slowly row by row. Comparing variables between the two debug sessions the errors in the C code where possible to find.

A problem occurred as the output string of exponents differs between the FORTRAN and C. For instance the output of “1000” will result as “1e003” in C and as “1e03” in FORTRAN. Hence a new output function was needed that writes 1000 as 1e03 in C. Instead of the function `fprintf()` the function `F77fprintf()` were created which works exactly the same way except for the exponential part.

Simulating even a small number of particles creates huge log files. Normal log files were between 30-200Mb in sizes. The log files could contain the particles energy, position, step length, deposited energy, type of interaction, interaction counter and so on for each interaction. Even so, comparing these large files row by row only took a couple of seconds. An external program was written to do the line comparisons. When two lines did not match between the log files the program printed which line together with the data. Hence, it was known if particle deposited the wrong energy at the n’th interaction etc.

It is important that the log files are identical in structure otherwise it would not be possible to compare line by line. Therefore the compiler OpenWatcom (OW) where used. OW is an open source version of the old compiler Watcom. OW contains both a C/C++ compiler and as well a FORTRAN compiler. This makes it possible to compile both program with OW and thus they would use the same mathematical libraries. If the FORTRAN and C code where to use different math libraries then mathematical expressions such as exp(), sin() and sqrt() would be evaluated differently. The decimals would not be the same for the same variables in the two codes even though the codes were the same. Thus, it would not be possible to run the line by line comparisons.

Another problem that occurred with decimals errors in the log where much harder to find and remedy. By running the debugger the problem where found. FORTRAN does not support the += operator which C does. Therefore the expression \( k = k + expr \) could be written as \( k += expr \) in C but not in FORTRAN. Since C uses this operator all of these expression where transformed in to the C variant. But when performing the log comparisons they differed due to this. The reason for it is that the right hand side is evaluated differently in some cases because of evaluation order. The few places where it occurred where replaced by the FORTRAN style.

4.4 Optimizations

During the code implementation, considerations were taken to improve the speed of the code. Mentioned above is the removal of the goto-calls.

Another improvement was to remove some unnecessary variables without changing the code. Sometimes a code block uses two different variables when performing a calculation and they are not needed at the same time. If you are able to successfully remove one of them
without changing the final result the code might run faster. That is because the code optimizer might produce code that needs less registers and hence the CPU needs to fetch data less often. This can yield some speedup depending on how smart the code optimizer is.

The new Penelope supports a variable number of materials, leading to a slower execution of the code. The reason is that when you access a dynamic variable two addresses is needed instead of one. When you use a fixed number of materials the address are at fixed positions. Then only one address is needed plus one offset value that is fixed. This lets the compiler decide the address directly and thus only one fetch from the memory is needed.

### 4.4.1 The Random function

Running PenelopeC trough a profiler showed that the Rand function, which produces random values between 0 and 1, took up ~40% of the CPU time. The main reason for this is that the Rand function consists of a lot of large integer divisions and it is being called many times.

In an attempt to reduce the CPU time for the Rand function a new version where created. The new version should of course produce exactly the same random numbers. It creates a buffer with \(n\) random numbers the first time it is being called. Then it returns one value at a time until it runs out of them in the same order as in the old version. When it is out of numbers it creates \(n\) new ones.

This reduced the amount of CPU time spent in the Rand function to ~20% compared to the old version. The test was performed with the simplest possible geometry in order for the simulation to spend as little CPU time as possible with geometry calculations. The geometry consisted of a simple slab, which is two infinite parallel planes defining a region of material. This has to be considered a successful implementation.

### 4.4.2 The secondary particles stack procedure

When a secondary particle is created it is being pushed on a stack so that the current particle can be simulated further. When the current particle is annihilated the program pops the secondary particle from the stack and starts simulating it.

In order to improve the secondary particle stack an attempt were made by using better memory management. The profiler did not show any significant improvement. Normally in a simulation the stack only fills with 5-10 secondary particles. However some variance reduction techniques produce a large number of secondary particles in the stack. One interaction could produce as much as 100-1000 such particles, and the new techniques will in this case lead to increased performance. Unfortunately the current version of PenelopeC does not support this kind of variance reduction techniques and that test could not be performed.

### 4.5 Results

After using the verification tool to see that the new C code simulated exactly the same as the FORTRAN code the program where put to the test. It performed the calculations about 20% faster compared to the FORTRAN code with minimal geometry.

### 4.6 Distribution

When PenelopeC was completely ported an interesting extension was implemented. Since the Monte Carlo simulation problem is very easy to parallelize PenelopeC was extended with distribution code in order to split the job between computers. The idea was to make it available for persons that do not have a cluster but only an office net, for example hospitals,
universities and other persons interested in Monte Carlo simulations. The code is based on a distribution package made for FreeNet by Jan Danielsson at Treadstone HB.

The MC problem does not require a lot of communication during the simulation. It is only under the initialization and when the simulation is done that most communication is required. Since a simulation could run for days many people might be interested in running the simulation on say 10 computers instead of one and thus reducing the computation time with approximately a factor of ten.

4.6.1 Structure

The structure of the distribution is to use one master computer and multiple slave computers, as shown in Figure 4-2.

![Figure 4-2: Example of a network distribution.](image)

The master computer is responsible for initiating the simulation, and making sure that everything is ready to be simulated, i.e. it is initializing the material and geometry before trying to contact any slaves. The master has a list of IP-numbers or host names which it uses to establish a connection via TCP/IP sockets. Once a connection is established between the master and a slave the simulation data is sent to the slave. Then the slave initializes the material and geometry notifying the master when done.

When all the slaves have been contacted the master has a list of the slaves it managed to establish a connection with. The master starts to simulate itself and for each new primary particle it listens if any slave is available. If the master sees an available slave it sends a message telling the slave to simulate a certain number of particles. When the slave has simulated all those particles it lets the master know it is available again.

The master keeps track of how many particles each slave has simulated, and in the case of a slave is lost the master redistribute the number of particles the slave had simulated. Thus, the master insures that the correct number of particles is simulated.

4.6.2 Slave considerations

When installing a slave on a computer several steps of precaution have to be taken, especially if users are working with other things on the computers during the simulation. It is important that the users do not have to be aware of the program running in the background. This can be achieved by lowering the priority of the process and making sure that the simulation does not require too much memory. As the MC simulation does not require much communication, the network traffic will not be stressed.

During the initialization the master sends the simulation files to the slaves and this could be a potential security threat. Someone could send a new file that overwrites an important operating system file. However during the file transfer the file names are truncated so that they only consist of the file name and not the complete path. This makes sure that the file is stored in the slave current directory.
4.6.3 Distribution testing

Simulations on the Elekta office network and cluster have been made. The cluster consists of 23 computer nodes and we used up to 10-12 computers on the office network. The speed of the computers where around 2-3GHz. Some of the office computers had two CPU’s and thus two slaves could be used simultaneously on those.

Few simulations where performed on the cluster but they performed well. This is to be expected since the cluster nodes do not have a lot of other processes running at the same time. The office machines, which used Windows, were slightly faster than the cluster machines. However, since they had a lot more processes running at the same time and the fact that people worked on them during the simulation made the Penelope process get less CPU time. Even so the office distribution simulated as fast as the cluster variant. This is because most of the time a person is working with a computer the computers full potential is not used. For instance, quite often the computer waits for the person to type something or click a mouse button. During this idle time the Penelope process gets CPU time and simulates particles.
5 GPU programming

5.1 History and Introduction

Graphics cards in ordinary PC’s have evolved dramatically during the last decade. The reason is the Gaming industry that demands faster and better graphics card in order to visualize more and more complex scenes in real time.

It all started in 94 when 3DFX released the first Voodoo I card. Before that, every game developer used the CPU when they implemented their 3D-engine. The 3D-engine is the graphics core of a game and is well suited to implement in a pipeline fashion. A simple 3D engine is provided in Figure 5-1.

![Figure 5-1: Simple 3D engine.](image)

Basically it takes a set of triangles processes them and then displays them on screen. The first step is to transform a triangle into different coordinate systems. This is applied to the triangle vertices, shown in Figure 5-2, and is called the transformation stage.

![Figure 5-2: Triangle with its corresponding vertices.](image)

The first part of the transformation stage is to transform the triangle from model space into scene space. This puts the triangle where it should be spatially located in the Euclidean space. Another transformation occur which puts the camera in origo, called the view transformation. The view transformation follows by the perspective transformation that makes triangles further away appear smaller.

The next part in the pipeline is clipping. The purpose of clipping is to remove the parts of the scene that is not visible to the viewer. This can of course be done in different stages in the pipeline. The aim with clipping is to reduce the amount of data sent through the pipeline. After the clipping has been performed the lighting calculation of the vertices is computed. The result from this stage is later used during the rasterization.

The final stage before the result is sent to the framebuffer is the rasterization stage. This stage uses the information from the lighting calculation and together with texture maps in order to draw the triangle, as it would appear on the screen.

The result is then sent to the framebuffer, which is the part in the computer’s memory that is displayed on the monitor.

Old games like Wolfenstein3D and Doom implemented this kind of 3D engine using only the CPU. That works, but games do not only consist of graphics. There is also Artificial Intelligence, collision detection, scene handling etc. If you could reduce the amount of CPU power spent on the graphics pipeline, more power would be available for other things and the overall game quality could be improved.

What 3DFX did in -94 was to implement some parts of the engine inside a graphic card and thus reducing the CPU computation time. They started by moving the transformation stage from the CPU to the GPU. Other features such as the z-buffer were also moved to the
graphics card. This was a tremendous breakthrough and the quality of games improved. Hence new graphics cards were created that moved more and more of the pipeline from the CPU to the GPU. The whole Texture&Lighting stage were implemented. The game industry demanded even more and eventually the whole pipeline was implemented on the graphics card.

Two different API’s exists for rendering graphics, OpenGL and DirectX. OpenGL has the advantage of being platform independent and DirectX is only used under Windows. DirectX on the other hand has some advantages since it knows what operating system it runs on. See Figure 5-3 for an overview of OpenGL rendering pipeline.

A fixed function pipeline is normally used as a state machine. That means that the programmer set certain states which describes how triangles will be processed through the pipeline. By changing the states the triangles will be rendered differently. One state could be whether the triangles should be rendered transparent another one could be whether flat or gouraud shading should be used. This limits the number of visual effects to what the vendor has made available on the card.

Eventually, companies realized that parts of the pipeline could be programmable on the graphics card. The reason for this was that a fixed function pipeline and certain number of effects were supported by the GPU, but with the programmability only the imagination was the limit of the effects that the GPU could produce. NVIDIA was the first company to release a programmable GPU on their Geforce3 card. This card had a programmable vertex unit (PVU) which could be programmed with up to 128 instructions. The PVU is usually called the Vertex Shader (VS). 128 instructions might not seem much but one have to bear in mind that a graphics card should be able to process a huge amount of triangles. If the VS would take too much time processing a triangle it would be a bottleneck in the pipeline and the scene could not contain a lot of triangles.

The VS replaces the transformation and lighting stage of vertices in the pipeline. With a VS the programmer can modify the transformation part exactly as he or she wishes. For example, he could perturb the vertices so that characters in games seem to have changing geometry (such as muscles contracting) or change the normal assigned at a vertex (creating bump-
mapping effects\(^4\). The bump-mapping techniques were already supported through bump-maps but now it could depend on vertex position instead of a look up from a bump-map. This would add more realism in games. The major breakthrough here was that it is up to the programmer to create these effects not being limited by a fixed function pipeline.

The rendering pipeline is easy to parallelize. Note that the triangles, in the pipeline are not dependent on one another. Eventually the graphics vendors implemented multiple PVUs on the card. Hence the GPU could process more than one vertex at the same time. The result was that more and more instructions were allowed in the VS program.

Now the graphics vendors looked at implementing a programmable pixel shader. This is not a trivial task because of the huge amount of pixels that is being rendered per frame. A triangle consists of only three vertices, but when rendered to the screen it usually consists of more than three pixels. Therefore the parallelization of the pipeline is used and the Pixel/Fragment shaders are implemented in parallel.

The need for speed created more and more nice and interesting features in GPU’s. One new machine instruction that calculates both the sinus and cosine value simultaneously, since both are usually needed, is an example. Another feature is swizzling that makes it possible to assign different vector elements to other elements. For instance: \(v_2.wzyx = v_1.xyzw\) rearrange the elements from \(v_1\) backwards in \(v_2\). This makes it possible to perform cross products fast.

The demand for doing more general calculations with GPU made extensions such as the P-buffer available. The P-buffer makes it possible to render to an off screen buffer or a texture. If you have a display device that can only display 8-bits colors but you need to render a picture using 32-bits color a P-buffer solves the problem. The P-buffer supports 128-bits color that enables rendering using single precision for each color component. Single precision means floating point values that uses 32-bits representation and the GPU follows the IEEE standard. This means that through the whole pipeline single precision values could be used and the output could be as well in single precision. That makes it possible to do more general calculation on the GPU instead of displaying triangles. See the chapter 5.3 for more information on general GPU programming.

With the GeForce-6 series, NVIDIA implemented one of the most important features for doing more general computation of the GPU and that was branching. Before the graphics vendors deemed it to time expensive to support branching, but now it was possible. The reason branching is time expensive is that it breaks the pipeline flow.

### 5.2 Programmable Vertex and Fragment shaders

In the beginning the VS and FS where programmed using assembly language. Instructions for general vector and matrix operations where implemented. Special instructions for vector reflection and refraction calculation and an instruction for calculating both the sinus and cosine value where also made available for further performance.

However, the need for a high-end language in order to make it easier to program the vertex and fragment shader was imminent. Therefore, different solutions where created. For DirectX the language HLSL where created which is almost identical to CG which was created by NVIDIA. For OpenGL the language GLSL was released.

CG is available on most platforms (Windows, Mac, and Linux) and you can create programs for both OpenGL and DirectX. A nice feature is that you can choose whether CG should compile the code each time you run your application or if you wish to compile only once. If you choose to compile each time you run the application, CG can take into account what GPU’s is on the current graphics card and optimize the generated code. Hence if a new

\(^4\) Bump-mapping – Changing the surface normal in order to make the surface appear bumpy without changing the geometry.
graphics card is installed, the program could utilize some of the new features in the card since the code is recompiled each time you run the application.

A simple vertex shader program is provided in Listing 5-1.

```c
struct appdata
{
  float4 hposition;
  float4 color;
};

struct vfconn
{
  float4 HPos;
  float4 Col0;
};

vfconn main(appdata IN, uniform float4x4 ModelViewProj)
{
  vfconn OUT = (vfconn) 0;
  OUT.hposition = mul(ModelViewProj, IN.hposition);
  OUT.Col0 = IN.color;
  return OUT;
}
```

Listing 5-1: A simple vertex shader program in Cg. Input to the program is a point in homogenous coordinates and a color. It also receives a reference to OpenGL’s ModelViewProjection matrix. It transforms the points with the matrix and assigns the color and passes them on to the fragment shader.

### 5.3 GPGPU

The speed of graphics card has improved dramatically over the recent years and they outperform the CPU. See Figure 5-4 for more details about the GPU vs. CPU performance. The amount of GFLOPS (Giga floating point operations per second) you can get from the GPU is not to be taken lightly. But can all this power be used to something else than just displaying graphics for computer games? The answer is yes. This is called GPGPU which stands for General Purpose computation on GPU.

Due to the limitations of the VS and FS not all problems can be computed on the GPU. But problems that can be implemented can get accelerated order of magnitudes.

What kind of computational problems are possible to move from the CPU on to the GPU? The most obvious might be filtering of a picture. By implementing a FS program with texture lookups it is fairly simple to implement this. However, most problems are not this trivial. But GPU vendors have made features available that makes it possible to do more general calculation. One example is the p-Buffer extension that makes it possible to render to a target that is not the framebuffer. Therefore one can render to a floating point buffer. This might be the most important extension for GPGPU.

Here is a list of some GPGPU application that has been achieved.

- GPU-Accelerated Computed Tomography
- Real-Time 3D Fluid Simulation
- Hardware Accelerated Ice Crystal Growth
- The Discrete Wavelet Transform on a GPU

ACM SIGGRAPH had 2004 a full day course in GPGPU for the first time. The evolution of the GPU compared with the CPU as can be seen in Figure 5-4.
5.4 Algorithm

The aim was to speed up the intersection between a ray and a triangle. In order to make things simple the algorithm was implemented in the vertex shader. However, the VS is not as fast as the FS. The reason is that the FS is used more often during the rendering process. Using the FS would require storing the triangles in textures, which could be a bit tedious. A speedup by a factor of 2 should be expected if the algorithm is implemented in the FS on a new GPU, as the number of parallel pipelines is higher in FS.

The first naïve attempt was to introduce branching in the code. See Appendix A for more information on the original ray/triangle intersection code. Obviously some sort rejection test has to take place in order to throw away triangles that do not intersect the ray.

The VS receives only one vertex at a time. Therefore something has to be done to feed the VS with data. Sending the vertices together with color and normal attributes solved the problem of sending the triangle data to the VS.

Figure 5-5: A vertex with normal and color attributes, which contain the triangle vertices.

Hence, by storing the coordinate of the first vertex in the vertex position, the second coordinate stored in the normal and the third in the color attribute, it is possible to send a
triangle to the VS. See Figure 5-5 for geometric interpretation. A bunch of triangles can be sent to the VS by using VertexArray, NormalArray and ColorArray in OpenGL. The triangles are still processed one at a time in the VS.

The z coordinate stores the intersection point, if there is no intersection it is left with a really large initial value. By manually setting the x and y coordinate for the triangle intersection to a fixed coordinate it is possible to read what the closest intersection was from the z-Buffer. This requires that the z-Buffer is turned on in OpenGL. The z-Buffer performs depth test on pixels when they are drawn. Hence, it is possible to let the z-Buffer perform the sorting of the closest intersection. This is of course done in the GPU as well.

```cpp
struct appdata
{
    float4 hposition; // Vertex 0
    float3 normal;    // Vertex 1
    float3 color;     // Vertex 2
};

struct vfconn
{
    float4 hposition;
    float3 color;
};

vfconn main(appdata IN,
uniform float3 ray_position,
uniform float3 ray_direction)
{
    vfconn OUT = (vfconn)0;
    float t = 1.0e5f, u, v;
    float3 edge1, edge2, tvec, pvec, qvec;
    float det, inv_det;
    const float Epsilon = 0.000001f;
    edge1 = IN.normal * IN.hposition.xyz;
    edge2 = IN.color * IN.hposition.xyz;
    pvec = cross(ray_direction, edge2);
    det = dot(edge1, pvec);
    if ( det > Epsilon || det < GEpsilon ) {
        inv_det = 1.0f / det;
        tvec = ray_position * IN.hposition.xyz;
        u = dot(tvec, pvec)*inv_det;
        if ( u >= 0.0f && u <= 1.0f ) {
            qvec = cross(tvec, edge1);
            v = dot(ray_direction, qvec)*inv_det;
            if ( v >= 0.0f && u+v <= 1.0f ) {
                t = dot(edge2, qvec)*inv_det;
            }
        }
    }
    OUT.hposition = float4(-1,-1,t,1);
    OUT.color = float4(1,1,1,1);
    return OUT;
}
```

Listing 5-2: First vertex shader program that calculates the intersection between a ray and a triangle. This program uses three different branches.

The problem with reading data from the GPU is that it is really slow. However, for this kind of algorithm that is of little importance, since we only read one pixel from the z-Buffer once. The new PCI Express bus will remedy this and it will provide faster read back from the GPU than the old AGP bus. See Listing 5-2 for the first VS code listing without optimizations.
5.5 Optimisation

During the testing of the algorithm it was discovered that the GPU algorithm could be optimised. It is actually possible to remove the three branches from the code and instead utilise the alpha test. The alpha value is used for drawing transparency objects. A color of a pixel is usually specified as \( <r, g, b, a> \) where \( r \) is red, \( g \) is green, \( b \) is blue and \( a \) is the alpha value. The alpha value describes the amount of background that is visible through the pixel. Hence setting the alpha value to some value that will discard the pixel in the alpha test will produce the same result as before. In Listing 5-3 is the code for the optimized version shown.

```
struct appdata
{
    float4 hposition; // Vertex 0
    float3 normal;    // Vertex 1
    float3 color;     // Vertex 2
};

struct vfconn
{
    float4 hposition;
    float3 color;
};

vfconn main(appdata IN,
            uniform float3 ray_position,
            uniform float3 ray_direction)
{
    vfconn OUT = (vfconn) 0;
    float t, u, v;
    float3 edge1, edge2, tvec, pvec, qvec;
    float det, inv_det;

    edge1 = IN.normal G IN.hposition.xyz;
    edge2 = IN.color G IN.hposition.xyz;
    pvec = cross(ray_direction, edge2);
    det = dot(edge1, pvec);
    float alpha = max(0, abs(det) G 0.00001) * 1e10;
    inv_det = 1.0f / det;
    tvec = ray_position G IN.hposition.xyz;
    u = dot(tvec, pvec)*inv_det;
    qvec = cross(tvec, edge1);
    v = dot(ray_direction, qvec)*inv_det;

    float2 uv = float2(u, v);
    float2 satuv = saturate(uv);
    float2 uvdiff = uv - satuv;
    alpha = dot(uvdiff, uvdiff) * 1e32;
    t = dot(edge2, qvec)*inv_det;

    OUT.hposition = float4(1,1,t,1);
    OUT.color = float4(1,1,0, alpha);
}
```

Listing 5-3: Second shader program that calculates the intersection between a ray and a triangle. This program uses no branches at all, instead the alpha test takes care of the discarding the triangles that do not intersect the ray.

5.6 Results

When comparing the algorithms against the CPU 100k triangles were intersected against 200 rays. A Pentium4 3.2GHz was used. The computer contained an ATI Radeon X800xt graphics card which contains 8 parallel vertex shaders.
The optimization of the vertex program yields a 27% faster computation. Comparing the GPU with the optimized version shows a ~800% speedup. This was expected since there is 8 vertex shaders present. Further optimization of the GPU code might provide additional speedup. The CPU version will be much harder to improve but this can be done with Streaming SIMD Extensions (SSE) optimization. But the theoretical limit of SSE optimization is only 400%.
6 Conclusions

If your aim is to speed up MonteCarlo simulations you might benefit from using and implementing parts of this thesis. Combining all the different methods described here into a single MC simulation package would provide a powerful tool. In order to create a really powerful simulation package it should be extended with both more variance reduction techniques and spatial subdivision (BSP-trees & Octtrees).

The GPU optimization technique presented here could be improved further and since the graphic card evolution seems to leave the CPU further behind this could provide very powerful computation for a low price. The drawback of having only single precision computation seems to be acceptable for this kind of problem.

If you would like to be able to produce a really powerful MC simulation package you have to support as many different platforms as possible while at the same time utilizing each computer’s individual computational capabilities.
7 Appendix A – A fast ray / triangle intersection algorithm.

7.1 Introduction
The algorithm, from Tomas Möller and Ben Trumbore, for computing the intersection between a ray and a triangle is very important in order to write fast code. The key issue is to use minimal amount of memory and still have an efficient algorithm. This method computes a vector \(<t, u, v>\) if the ray intersects the triangle. Here \(t\) is the distance from the ray origin and \(u\) and \(v\) are the bary-centric coordinates for the intersection.

This method is not as fast as Badouel’s algorithm, but that algorithm stores the plane equation for each triangle as opposed to this one which do not. Therefore a lot of memory is saved by using this algorithm since the only data that needs to be stored is the triangle vertices.

No preprocesing is required either in this algorithm.

7.2 Algorithm
A ray \(R(t)\) is defined as \(R(t) = O + tD\) where \(O\) is the origin and \(D\) is the direction of the ray. The triangle is defined by its three vertices \(V_0, V_1\) and \(V_2\). The intersection algorithm will yield the value of \(t\) where the ray intersects the triangle. The two values \(u\) and \(v\) are the barycentric coordinates where the intersection occurred. These could be used for texturing or normal interpolation etc.

A point on the triangle can be expressed as

\[ T(u, v) = (1 – u – v)V_0 + uV_1 + vV_2. \]

Equation 1

Where \(u\) and \(v\) must fulfill \(u \geq 0, v \geq 0\) and \(u + v \leq 1\). The \(u\) and \(v\) are barycentric coordinates.

Computing the intersection between the ray and the triangle is equivalent to solving the equation where \(R(t) = T(u, v)\). This can be expressed as: \(O + tD = (1 – u – v)V_0 + uV_1 + vV_2\). Writing this in matrix form yields:

\[
\begin{bmatrix}
-D & V_1 - V_0 & V_2 - V_0
\end{bmatrix}
\begin{bmatrix}
t \\
u \\
v
\end{bmatrix}
= O - V_0
\]

Equation 2

Finding the intersection between the ray and the triangle is equivalent to solving the linear system in Equation 2.

This could geometrically be viewed as transforming the triangle into a unit triangle in the yz-plane and where the ray is parallel with the x-axis. This is illustrated in Figure 7-1.
Denoting $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$ and $T = O - V_0$. The solution to our system is found by using Cramer’s rule.

$$\begin{vmatrix} t \\ u \\ v \end{vmatrix} = \frac{1}{D, E_1, E_2} \begin{vmatrix} T, E_1, E_2 \\ D, T, E_2 \\ D, E_1, T \end{vmatrix}$$

Equation 3

Since we have the following relation: $|A, B, C| = - (A \times C)B = -(C \times B)A$, we can rewrite Equation 3 as.

$$\begin{vmatrix} t \\ u \\ v \end{vmatrix} = \frac{1}{(D \times E_2)E_1} \begin{vmatrix} (T \times E_1)E_2 \\ (D \times E_2)T \\ (T \times E_1)D \end{vmatrix} = \frac{1}{PE_1} \begin{vmatrix} QE_2 \\ PT \\ QD \end{vmatrix}$$

Equation 4

Common factors from Equation 4 where $P = D \times E_2$ and $Q = T \times E_1$ are used in the computation in order to speed things up.

### 7.3 Code

Below in Listing 7-1 is the code used in Penelope to calculate intersections between particles and surfaces. The code implemented on the GPU is based on this code.

```c
// find vectors for two edges sharing the first vertex
Sub(edge1, triangles[c].v2, triangles[c].v1);
Sub(edge2, triangles[c].v3, triangles[c].v1);

// begin calculating determinant - also used to calculate u parameter
Cross(pvec, (*dir), edge2);

// if determinant is near zero, ray lies in plane of triangle
det = Dot(edge1, pvec);

// test if the ray is parallel with the triangle plane
if (det > EPSILON && det < EPSILON)
    return 0;

inv_det = RL(1.0) / det;

// calculate distance from the first vertex to ray origin
Sub(tvec, (*pos), triangles[c].v1);
```
// calculate U parameter and test bounds
u = Dot(tvec, pvec) * inv_det;
if (u < RL(0.0) || u > RL(1.0))
    return 0;

// prepare to test v parameter
Cross(qvec, tvec, edge1);

// calculate v parameter and test bounds
v = Dot(*dir, qvec) * inv_det;
if (v < RL(0.0) || u + v > RL(1.0))
    return 0;

// calculate t, where the ray intersects the triangle
t = Dot(edge2, qvec) * inv_det;
return 1;

Listing 7-1: Code listing that calculates the intersection between a ray and triangle.
8 References

Francesc Salvat, José M. Fernández-Varea
Penelope: A Code System for Monte Carlo
Simulation of Electron and Photon Transport

Tomas Möller and Ben Trumbore.
Fast, minimum storage ray-triangle intersection.

Didier Badouel
An Efficient Ray-Polygon Intersection
Graphics Gems (1990)

GPGPU Homepage
www.gpgpu.org/

ATI Homepage
www.ati.com/

NVIDIA Homepage
www.nvidia.com/

ShaderTech
www.shadertech.com/